МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

ИНТЕГРАЦИЯ БОЛЬШИХ ДАННЫХ В SQL SERVER: ИСПОЛЬЗОВАНИЕ СОЕДИНИТЕЛЯ APACHE SPARK ДЛЯ SQL SERVER

КУРСОВАЯ РАБОТА

студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Ларцева Александра Андреевича

Научный руководитель
старший преподаватель _____ М. И. Сафрончик

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В XXI веке самым ценным ресурсом стала информация, использование которой позволяет достичь новых высот во всех областях человеческой деятельности. Информации стало настолько много, что хранить и обрабатывать ее традиционными способами стало очень сложно, к тому же данные, обработанные традиционно, появляются, как правило, с опозданием, следовательно будут уже не актуальны. Кроме того, что различные организации создают огромные объемы данных, большая часть которых представлена в различных форматах (текстовые документы, видео файлы, машинные коды и т.д.), локализованных в разнообразных хранилищах, эти данные весьма часто обновляются. Термин «большие данные» (Big Data) существует уже почти двадцать лет, став за это время всемирно обсуждаемым. За историю своего недолгого существования он успел получить широкую популярность – с одной стороны эти технологии вызывают некоторый скепсис, с другой стороны большое число компаний уже внедрили Big Data в свою деятельность, что позволило оптимизировать работу с данными.

Цель данной работы изучение технологий Big Data и их взаимодействия с системами управления реляционными базами данных, на примере Microsoft SQL Server.

1 Технологии поиска, обработки и применения неструктурированной информации в больших объемах данных

1.1 Понятие Больших данных

Big Data – это новые возможности, ставшие доступными, благодаря новым технологиям и подходам к обработке больших объемов данных и использованию вычислительных мощностей. Хотя термин «большие данные» является относительно новым, процесс сбора и хранения больших объемов информации для последующего анализа имеет давнюю историю. Концепцию больших данных можно описать совокупностью факторов объема, скорость и разнородности данных:

Объем данных заключается в том ,что множество организаций собирают данные из различных источников, включая бизнес-транзакции, социальные сети и информацию от датчиков или машинных данных. В прошлом хранить его было бы проблемой, но новые технологии (такие как Hadoop) облегчили бремя. Скорость данных состоит в том ,что потоки данных происходят с беспрецедентной скоростью и должны обрабатываться своевременно. RFID-метки, датчики и интеллектуальный учет управляют необходимостью иметь дело с потоками данных в близком к реальному времени. Разнородность данных следует из того ,что данные поступают во всех типах форматов – от структурированных, числовых данных в традиционных базах данных до неструктурированных текстовых документов, электронной почты, видео, аудио, биржевых данных и финансовых транзакций. Так же к предыдущем свойствам можно добавить еще изменчивость и сложность данных: Под изменчивостью имеется ввиду, что дополнение к возрастающим скоростям и разновидностям данных потоки данных могут быть сильно несовместимы с периодическими пиками. Ежедневные, сезонные и вызванные событиями пиковые нагрузки данных сложны в обработке. Тем более при работе с неструктурированными данными. Сложность данных подразумевает под собой то ,что в наши дни данные поступают из нескольких источников, что затрудняет связь, сопоставление, очистку и преобразование данных между системами. Однако необходимо соединять и коррелировать связи, иерархии и множественные связи данных, иначе данные могут быстро выйти из-под контроля. [1] Сами по себе алгоритмы Big Data возникли при внедрении первых высокопроизводительных серверов (мэйнфреймов), обладающих достаточными ресурсами для

оперативной обработки информации и пригодных для компьютерных вычислений и для дальнейшего анализа. Сам термин Big Data впервые был озвучен в 2008 году на страницах спецвыпуска журнала Nature в статье главного редактора Клиффорда Линча. [2] Big Data – серия подходов, инструментов и методов обработки структурированных и неструктурированных данных огромных объёмов и значительного многообразия. Данные технологии применяются для получения воспринимаемых человеком результатов, эффективных в условиях непрерывного прироста, распределения информации по многочисленным узлам вычислительной сети. Исходя из определения Big Data, можно сформулировать основные принципы работы с такими данными: Горизонтальная масштабируемость. Поскольку данных может быть сколь угодно много – любая система, которая подразумевает обработку больших данных, должна быть расширяемой. Отказоустойчивость. Принцип горизонтальной масштабируемости подразумевает, что машин в кластере может быть много. Следовательно, часть машин будут выходить из строя, поэтому методика работы с большими данными должны учитывать возможность таких сбоев. Локальность данных. Это один из важнейших принципов проектирования Big Data – решений является принцип локальности данных – по возможности обрабатываем данные на той же машине, на которой их храним. [3] Традиционным методом работы с массивами информации являются реляционные базы данных. Однако работа с реляционной базой данных на сотни терабайт - это еще не Big Data. В таблице (см. таблицу 1) представлены характеристики и из отличия в традиционных БД и Big Data.

Таблица 1 – Характеристика больших БД и реляционных БД

Характеристика	Реляционные БД	Большие БД
Объем хранимой информации	От 1гб до 10тб	От 1pb до 1eb
Способ хранения	Централизованный	Распределенный
Структурированность данных	Структурирована	Полуструкурированна или неструктурированна
Взаимосвзяь данных	Сильная	Слабая

В реляционных БД информация распределена дисперсионно, т.е. имеет место изначально заданная четкая структура, изменение которой в уже работающей базе связано с множеством проблем. Таким образом, в силу своей архитектуры, реляционные БД лучше всего подходят для коротких быстрых

запросов, идущих однотипным потоком. Сложный же запрос либо потребует перестройки структуры БД, либо, в угоду быстродействию, увеличения вычислительных мощностей. Это указывает на еще одну проблему традиционных баз данных, а именно на сложность их масштабируемости. Таким образом, для работы со сложными гибкими запросами необходима среда, позволяющая хранить и обрабатывать неструктурированные данные, поддающаяся масштабированию и допускающая применения распределенных вычислений, где для обработки данных используется не одна высокопроизводительная машина, а целая группа таких машин, объединенных в кластер. [4]

1.2 Не реляционные модели данных, понятие NoSQL

Термин «NoSQL» возник в июне 2009 года и был расшифрован как «Not Only SQL» – «не только SQL». Таким термином обозначают не реляционные БД, в которых нет внутренних связей. БД NoSQL могут использовать различные модели представления данных в зависимости от своего назначения. Технология NoSQL убирает все ограничения реляционной модели (например, трудоемкость горизонтального масштабирования, недостаток производительности в кластере), а также облегчает способы хранения и доступа к данным. Такие БД используют неструктурированный подход, организуя данные специфических типов за малый промежуток времени и предлагая различные типы доступа к ним. [5] NoSQL база данных предоставляет механизм для хранения и извлечения данных, который моделируется средствами, отличными от табличных отношений, используемых в реляционных базах данных. Такие базы данных существовали с конца 1960-х годов, но не получили прозвище "NoSQL"до всплеска популярности в начале двадцать первого века, вызванного потребностями компаний Web 2.0, таких как Facebook, Google и Amazon. базы данных NoSQL все чаще используются в больших данных и веб-приложениях реального времени. системы NoSQL также иногда называют "Not only SQL чтобы подчеркнуть, что они могут поддерживать SQL-подобные языки запросов. [6]

Вместо этого большинство баз данных NoSQL предлагают концепцию "окончательной согласованности в которой изменения базы данных распространяются на все узлы "в конечном счете"(обычно в течение миллисекунд), поэтому запросы данных могут не возвращать обновленные данные немедленно или могут привести к чтению данных, которые не являются точными, проблема, известная как устаревшие чтения. Кроме того, некоторые системы

NoSQL могут демонстрировать потерянные записи и другие формы потери данных. Некоторые системы NoSQL предоставляют такие понятия, как ведение журнала с опережением записи, чтобы избежать потери данных. Для распределенной обработки транзакций в нескольких базах данных, согласованность данных является еще более сложной задачей, которая является сложной как для NoSQL, так и для реляционных баз данных. Даже текущие реляционные базы данных не позволяют ограничениям ссылочной целостности охватывать базы данных. Существует мало систем, которые поддерживают как транзакции ACID, так и стандарты x/Open XA для распределенной обработки транзакций. [7]

Базы данных NoSQL-это нереляционные базы данных, которые являются масштабируемыми, оптимизированными для использования моделей данных без единой схемы. Базы данных NoSQL широко используются, потому что они упрощают разработку, обеспечивают отказоустойчивость и обеспечивают низкую задержку. Такие базы данных могут использовать различные модели данных, включая столбчатые, документальные, графические данные и хранить пары "ключ-значение"в памяти. Системы базы данных NoSQL используются для управления данными различных моделей, в том числе для хранения пар "ключ-значение"в памяти графовых моделей данных и хранения документов. Эти типы баз данных оптимизированы для приложений, которым требуются большие объемы данных, низкая задержка и гибкие модели данных. Все это достигается за счет смягчения жестких требований к согласованности традиционных реляционных баз данных. Базы данных NoSQL отлично подходят для приложений, требующих более высокой масштабируемости и меньшего времени отклика, чем традиционные реляционные базы данных. Среди них много больших приложений для передачи данных, мобильных и интернет-приложений. Благодаря использованию упрощенных структур данных и горизонтального масштабирования базы данных NoSQL обычно обеспечивают меньшее время отклика, чем реляционные базы данных, и их легче масштабировать. Как нереляционные базы данных (NoSQL), так и реляционные системы управления базами данных (РСУБД) имеют свои преимущества и недостатки. СУБД позволяют гибко запрашивать данные, но запросы являются довольно дорогими и плохо масштабируемыми при увеличении трафика. Базы данных NoSQL позволяют эффективно запрашивать данные ограниченным числом

способов. Все остальные методы, как правило, дороги и имеют низкую скорость выполнения. [8]

Существует четыре популярных типа баз данных NoSQL: Базы данных столбцов оптимизированы для чтения и записи данных в виде столбцов, а не строк. Столбчатый подход к хранению таблиц базы данных важен для производительности аналитических запросов, поскольку он значительно снижает Общие требования к дисковому вводу-выводу и объем данных, которые необходимо загрузить с диска. База данных документов предназначена для хранения полуструктурированных данных в документах, обычно JSON или XML. В отличие от традиционных реляционных баз данных, каждый из документов NoSQL может иметь свою собственную схему, которая обеспечивает большую гибкость в организации и хранении данных приложения и уменьшает объем хранилища для необязательных значений. Графовые базы данных хранят вершины и направленные связи, называемые ребрами. Графовыми могут быть и базы данных SQL, и базы данных NoSQL. Каждая вершина или ребро может обладать набором собственных свойств. Хранилище типа «ключ-значение» - это база данных NoSQL, предназначенная для нагрузок с большим количеством операций чтения или нагрузки с высокими требованиями к вычислительной мощности. [9]

Преимущества NoSQL:

— Возможности хранения больших объемов неструктурированной информации.
— NoSQL базы лучше поддаются масштабированию, хотя масштабирование поддерживается и в SQL-базах, это требует гораздо больших затрат человеческих и аппаратных ресурсов.
— Горизонтальное масштабирование (несколько независимых машин соединяются вместе, и каждая из них обрабатывает свою часть запросов) позволяет увеличить емкость кластера за счет добавления нового сервера. Предназначенное для работы в распределенных системах NoSQL, хранилище спроектировано таким образом, чтобы все процедуры распределения данных и отказоустойчивости выполнялись базой данных NoSQL.

Ключевые преимущества NoSQL баз в распределенных системах заключаются в процедурах шаринга и репликации. Репликация — это копирование

обновленных данных на другие сервера. Это позволяет добиться большей отказоустойчивости и масштабируемости системы. Шаринг — это разделение информации по разным узлам сети. Каждый узел отвечает только за определенный набор данных и обрабатывает запросы, относящиеся только к этому набору данных. NoSQL предполагает, что шаринг реализует сама база данных, он будет производиться автоматически.

Недостатки NoSQL:

— Приложение сильно привязывается к конкретной СУБД.
— Ограниченная емкость встроенного языка запросов.
— Трудности быстрого перехода с одной нереляционной базы данных на другу. [10]

Различие между этими базами данных заключается главным образом в способах хранения и структурирования данных, а также доступа к ним. Разные типы баз данных оптимизированы для разных приложений и примеров использования. MapReduce – это модель программирования и связанная с ней реализация для обработки и создания больших наборов данных с параллельным распределенным алгоритмом в кластерах. MapReduce предполагает, что данные организованы в виде некоторых записей. Обработка данных происходит в 3 стадии. Обработка данных в фреймворке MapReduce производится в три шага:

— Map: каждый рабочий узел применяет функцию map к локальным данным и записывает Выходные данные во временное хранилище. Главный узел обеспечивает обработку только одной копии избыточных входных данных.
— Shuffle: рабочие узлы перераспределяют данные на основе выходных ключей (созданных функцией map) таким образом, чтобы все данные, принадлежащие одному ключу, находились на одном рабочем узле.
— Reduce: рабочие узлы теперь обрабатывают каждую группу выходных данных по ключу параллельно.

Плюсы и минусы технологии MapReduce: Все запуски функции map работают независимо и могут работать параллельно, в том числе на разных машинах кластера. Все запуски функции reduce работают независимо и могут работать параллельно, в том числе на разных машинах кластера. Shuffle внутри себя представляет параллельную сортировку, поэтому также может работать

на разных машинах кластера. Пункты 1-3 позволяют выполнить принцип горизонтальной масштабируемости. Функция map, как правило, применяется на той же машине, на которой хранятся данные – это позволяет снизить передачу данных по сети (принцип локальности данных). MapReduce – это всегда полное сканирование данных, никаких индексов нет. Это означает, что MapReduce плохо применим, когда ответ требуется очень быстро. [11]

1.3 Hadoop

Hadoop – представляет собой набор утилит с открытым исходным кодом, которые облегчают использование сети из многих компьютеров для решения проблем, связанных с большими объемами данных и вычислений. Он обеспечивает программную основу для распределенного хранения и обработки больших данных с использованием модели программирования MapReduce. Все модули в Hadoop разработаны с фундаментальным предположением, что аппаратные сбои являются общими вхождениями и должны автоматически обрабатываться платформой. Платформа Hadoop позволяет сократить время на обработку и подготовку данных, расширяет возможности по анализу, позволяет оперировать новой информацией и неструктурированными данными. Главные задачи платформы Hadoop — хранение, обработка и управление данными. Базовая платформа Hadoop состоит из следующих модулей:

- Hadoop Common - содержит библиотеки и утилиты, необходимые другим модулям Hadoop.
- Hadoop Distributed File System (HDFS) - распределенная файловая система, которая хранит данные на обычных машинах, обеспечивая очень высокую совокупную пропускную способность кластера.
- HADOOP YARN – введенная в 2012 году Платформа, отвечающая за управление вычислительными ресурсами в кластерах и их использование для планирования приложений пользователей.
- Hadoop MapReduce - реализация модели программирования MapReduce для крупномасштабной обработки данных. Решения, построенные на базе технологии Hadoop, обладают рядом существенных преимуществ. Основные из них приведены в таблице (см.таблицу 2). [12, 13]

Таблица 2 – Характеристика больших БД и реляционных БД

Приемущество	Краткое описание
Снижение времени на обработку данных	При обработке данных на кластере можно существенно сократить время на обработку данных
Повышение отказоустойчивости	Выход из строя одного или нескольких узлов кластера влияет только на производительность системы, при этом система продолжает корректно работать и предоставлять сервис конечным пользователям
Линейная масштабируемость	Решение позволяет наращивать производительность просто за счет добавления новых узлов кластера. При этом производительность кластера возрастает линейно
Работа с неструктурированными данными	Технология позволяет осуществлять сложную обработку любых файлов, в том числе неструктурированных, благодаря чему такие данные могут быть эффективно обработаны и использованы

2 Описание программных средств, используемых в данной работе

2.1 Spark

Apache Spark – это Big Data фреймворк с открытым исходным кодом для распределённой пакетной и потоковой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop Spark предоставляет быструю и универсальную платформу для обработки данных. По сравнению с Hadoop Spark ускоряет работу программ в памяти более чем в 100 раз, а на диске – более чем в 10 раз (см таблицу 3). Spark может работать

Таблица 3 – Отличия Spark и Hadoop MapReduce

Hadoop MapReduce	Spark
Быстрый	В сотни раз быстрее
Пакетная обработка данных	Обработка данных в реальном времени
Хранит данные на диске	Хранит данные в оперативной памяти
Написан на Java	Написан на Scala

как в среде кластера Hadoop под управлением YARN, так и без компонентов ядра хадуп, так же он поддерживает несколько языков програмирования таких как Java , Scala и Python. В Spark вводится концепция RDD (устойчивый распределенный набор данных) – неизменяемая отказоустойчивая распределенная коллекция объектов, которые можно обрабатывать параллельно. В RDD могут содержаться объекты любых типов; RDD создается путем загрузки внешнего набора данных или распределения коллекции из основной программы (driver program). В RDD поддерживаются операции двух типов:

- Трансформации – это операции (например, отображение, фильтрация, объединение и т.д.), совершаемые над RDD; результатом трансформации становится новый RDD, содержащий ее результат.
- Действия – это операции (например, редукция, подсчет и т.д.), возвращающие значение, получаемое в результате некоторых вычислений в RDD. [14]

Трансформации в спарк вычесляются по концепции ленивых вычеслений, что значит что результат не исчисляется до тех пор пока он не понадобиться. [15, 16] В Spark так же доступна структура DataFrame. Она представляет двумерную маркированную структуру данных. Визуально выглядит как таблица. DataFrame можно сформировать на основе RDD, для удобной работы с данными по средствам доступных функций группировки и выборки дан-

ных. [17]

2.2 SparkNLP

Spark NLP – это библиотека обработки естественного языка на Scala, Python и Java с открытым исходным кодом. Построена на основе Apache Spark и Spark ML. Цель Spark NLP обеспечить API с конвейерами обработки естественного языка. Конвеером является набор текстовых анотаторов, например таких как токенизатор, лематизатор и POS-теггер. [18, 19]

2.3 Соединитель Apache Spark для SQL Server и его использование

Соединитель Apache Spark для SQL Server - высокопроизводительный соединитель который позволяет использовать любую базу данных SQLв качестве источника или приемника данных. [20].Будет использоваться в работе для передачи данных в SQL Server.

2.4 Microsoft SQL Server

Microsoft SQL Server — СУБД, разработанная компанией Microsoft, в основном использующей язык запросов Transact-SQL. Особенностями данной СУБД является относительно простой интерфейс, синхронизация с другими базами данных, поддержка OLAP (компонент Analysis Services).SQL Server является основой платформы обработки данных Microsoft.В настоящей работе СУБД будет использованна, для приема данных после обработки ApacheSpark. [21, 22]

2.5 Microsoft Power Bi

Microsoft Power BI — это коллекция программных служб, приложений и соединителей, которые взаимодействуют друг с другом, чтобы превратить разрозненные источники данных в согласованные, визуально иммерсивные и интерактивные аналитические сведения. Power Bi может брать данные из SQL Server, для создания отчетов, что и будет использовано в данной работе. [23]

3 Реализация примеров

3.1 Скрипты на Scala, и загрузка данных в SQL Server

```
import org.apache.spark.sql.SparkSession

object Connector_demo {
    def main (args: Array[String]) :Unit = {
      val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
      val database_name = "Spark_base"
      val url =
            server_name + ";" + "databaseName=" + database_name + ";"

      val table_name = "New"
      val username = "sa"
      val password = "user"
      val logFile =
                "C:/MyFiles/Spark/spark-3.1.2-bin-hadoop3.2/README.md"

      val spark = SparkSession.builder.appName("Simple Application")
                                      .getOrCreate()
      val columns = Seq("language","users_count")
      val data = Seq(("Java", "20000"),
                     ("Python", "100000"),
                     ("Scala", "3000"))
      val rdd = spark.sparkContext.parallelize(data)
      val df1 = spark.createDataFrame(rdd)
      val dfRe = df1.toDF(columns : _*)
      dfRe.show()
      dfRe.write
      .format("jdbc")
      .mode("overwrite")
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", table_name)
      .option("user", username)
```

```
      .option("password", password)
      .save()
  }
}
```

Первый пример демонстрирует инициализацию подключения и запись в базу данных SQLServer, и создание простого RDD. В начале скрипта обьявляются текстовые переменные содержащие данные об имени пользователя, пароле , имени базы данных и сервера экземляра SQLServer. Далее инициализируется `SparkSession`. Переменная `columns` , отвечает за имя столбцов в будущей таблице SQL Server. После задается последовательность кортежей, которая превращается в RDD при помощи функции parallelize. Функция CreateDataFrame , превращает RDD в dataFrame. По умолчания имена колонок имеют значения `_1` и `_2`. Поэтому с помощью функции `.toDF`, имена колонок изменяются на имена указанные в переменной `columns`. После чего с помощью функции, `.write` данные записываются в таблицу SQLServer (см.рис 1). Параметр `overvrite` означает ,что существующая таблица будет перезаписана или , если ее не существует будет создана новая с указанным именем и параметрами. Параметры `format` и `driver`, необходимы для того что бы указать , что подключение будет установлено с помощью конектора.

```
import org.apache.spark.sql.SparkSession

object Connector_demo {
  def main (args: Array[String]) :Unit = {
    val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
    val database_name = "Spark_base"
    val url =
          server_name + ";" + "databaseName=" + database_name + ";"

    val table_name = "New"
    val username = "sa"
    val password = "user"
    val logFile =
              "C:/MyFiles/Spark/spark-3.1.2-bin-hadoop3.2/README.md"
```

```
        val spark = SparkSession.builder.appName("Simple Application")
                                        .getOrCreate()
        val columns = Seq("language","users_count")
        val data = Seq(("Java", "20000"),
                       ("Python", "100000"),
                       ("Scala", "3000"))
        val rdd = spark.sparkContext.parallelize(data)
        val df1 = spark.createDataFrame(rdd)
        val dfRe = df1.toDF(columns : _*)
        dfRe.show()
        dfRe.write
        .format("jdbc")
        .mode("overwrite")
        .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
        .option("url", url)
        .option("dbtable", table_name)
        .option("user", username)
        .option("password", password)
        .save()
    }
}
```

Results | Messages

	language	users_count
1	Java	20000
2	Python	100000
3	Scala	3000

Рисунок 1 – Новая таблица с именем New в базе данных

Второй пример отличается тем, что RDD создается на основе небольшого текстовго файла. И демонстрируется работа функциии `filter`. После инициализации SparkSession, используются функции `read.textFile`, для того что бы прочитать текстовый файл в RDD. После этого используются фукции filter для того чтобы найти строки содержащие буквы A и B. В результате

получается 2 датасета, на основе которых создаются новые датафрэймы с помощью функции .withColumn, и объединяются их по общему столбцу id в новый датафрейм. Переименовываются столбцы с помощью функции .toDF и записываются в таблицу SQLServer (см. рисунок 2).

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{monotonically_increasing_id}
object A_B {
    def main (args: Array[String]) : Unit = {
      val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
      val database_name = "Spark_base"
      val url =
            server_name + ";" + "databaseName=" + database_name + ";"
      val table_name = "As_and_Bs"
      val username = "sa"
      val password = "user"
      val logFile =
                "C:/MyFiles/Spark/spark-3.1.2-bin-hadoop3.2/README.md"

      val spark = SparkSession.builder.appName("A_B").getOrCreate()
      val logData = spark.read.textFile(logFile).cache()
      val As = logData.filter(line => line.contains("a"))
      val Bs = logData.filter(line => line.contains("b"))
      val A_1 = As.withColumn("id", monotonically_increasing_id)
      val B_1 = Bs.withColumn("id", monotonically_increasing_id)
      val data = A_1.join(B_1, Seq("id")).drop("id")
      val columns = Seq("A_word_line", "B_word_line")
      val dfRe = data.toDF(columns : _*)
      dfRe.write
        .format("jdbc")
        .mode("overwrite") /*overwrite*/
        .option("driver",
                "com.microsoft.sqlserver.jdbc.SQLServerDriver")
        .option("url", url)
        .option("dbtable", table_name)
```

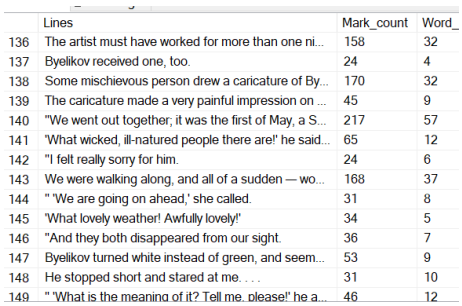
```
            .option("user", username)
            .option("password", password)
            .save()
        spark.stop()

    }
}
```


	A_word_line	B_word_line
1	# Apache Spark	MLlib for machine learning, GraphX for graph pro...
2	Spark is a unified analytics engine for large-scale ...	[![Jenkins Build](https://amplab.cs.berkeley.edu/je...
3	high-level APIs in Scala, Java, Python, and R, and ...	[![PySpark Coverage](https://img.shields.io/badge/...
4	supports general computation graphs for data ana...	guide, on the [project web page](https://spark.apa...
5	rich set of higher-level tools including Spark SQL f...	This README file only contains basic setup instr...
6	MLlib for machine learning, GraphX for graph proc...	Spark is built using [Apache Maven](https://mave...
7	and Structured Streaming for stream processing.	To build Spark and its example programs, run:
8	<https://spark.apache.org/>	./build/mvn -DskipTests clean package

Рисунок 2 – Новая таблица с именем A_B

Третий пример показывает работу с относительно большим текстовым файлом. Для демонстрации использовался текст повести А.П Чехова «Человек в футляре» на английском языке. После прочтения текста в RDD, датасет очищается от пустых строк с помощью функции `filter` и превращается в датафрейм с дополнительным столбцом `id`. Далее датафрейм без пустых строк превращается в таблицы содержащие данные о количестве слов и символов в каждой строке. Что бы посчитать количество символов, используется функция `map` с помощью которой заменяются пробельные символы в строке, с помощью функции `replace]`, и считается количество знаков с помощью функции `length`. Количество слов расщитываеться дважды с применением функции map, в первый раз , для того чтобы разделить строки на слова, путем удаления пунктуации ,с помощью регулярных выражений, и разделения строки через пробельные символы функцией `split`.Второй раз, что бы посчитать их количество. Потом из полученных датасетов создаются датафреймы с добавочным столбцом `id`, и соединяются функцией join. После чего добавочный столбец опускаю и загружаю датафрейм в SQLServer (см. рисунок3). [20]

Четвертый пример реализован на основе того же текстового файла. Характерен тем что в нем используются базовые модели Spark NLP. Сначало инициализируются модели DocumentAssembler, SentenceDetector и Tokenizer. DocumentAssembler, принимает на вход данные датасета и добавляет к нему столбец document. Столбец document, содержит в себе массив параметров, например начало и конец элемента считанного из датасета, и его значения. Так



	Lines	Mark_count	Word_
136	The artist must have worked for more than one ni...	158	32
137	Byelikov received one, too.	24	4
138	Some mischievous person drew a caricature of By...	170	32
139	The caricature made a very painful impression on ...	45	9
140	"We went out together; it was the first of May, a S...	217	57
141	'What wicked, ill-natured people there are!' he said...	65	12
142	"I felt really sorry for him.	24	6
143	We were walking along, and all of a sudden — wo...	168	37
144	" 'We are going on ahead,' she called.	31	8
145	'What lovely weather! Awfully lovely!'	34	5
146	"And they both disappeared from our sight.	36	7
147	Byelikov turned white instead of green, and seem...	53	9
148	He stopped short and stared at me. . . .	31	10
149	" 'What is the meaning of it? Tell me, please!' he a...	46	12

Рисунок 3 – Новая таблица man_in_the_case

же он содержит параметры необходимые на вход другим моделям, которые будут заполнять их по мере использования. SentenceDetector, берет на вход столбец document и возвращает столбец sentence, где возвращает метки на начало и конец предложений. Tokenizer, делит предложения на токены (слова или знаки препинания). После чего определяется Normalizer, он принимает на вход токены и в соотвествии в данными указаными в функции .setCleanupPatterns, удаляет знаки препинания, так же отставляя при этом численные индефикторы. В соотвествии с функцией .setLowerCase, Tokenizer приводит все токены в нижний регистр. Далее определяется Pipline , модель которые объединяет все предыдущие определения ,с помощью функции .setStages, в одно целое, и приведет их в исполнение по порядку. Далее имени result присваивается датафрейм, который вернет функция пайплайна transform. В каждой ячейке датафрейма находится масив токенов, поэтому с помощью функции explode строки разделяются на несколько строк, которые содержат масив состоящий из одного токена. После этого применяется функция map, внутри которой к каждой строке датафрейма применяется функция getString, которая извлекает из первого элемента строк, токен. Дальше в перменную hund записывается, количество строк в датафрейме и приводиться к типу Float. С помощью функции groupBy, датафрейм группируется по группам в соотвествии с уникальными значениями столбца value. С помощью функции count, формируется датафрейм состоящий из столбца уникальных значений,и их количества в фрейме данных. В переменную col_per записывается результат применения функций map, внутри которой рассчитвается процент который занимает каждое уникальное значений от количества всех нормализованных токенов, и withColumn, которая с помощью функции monotonically_increasing_id. Добавляет столбец id, который будет позже использован ,для объединения с другим датафреймом. В переменную word_frequency будет записан результат обьединения датафрей-

ма содержащегося в переменной col_per, и датафрейма dif_w_count. Значение перемнной word_frequency, датафрейм который будет передан в SQL Server.

```
import com.johnsnowlabs.nlp.annotator._
import org.apache.spark.sql.functions._
import com.johnsnowlabs.nlp.{SparkNLP, annotators}
import com.johnsnowlabs.nlp.base.DocumentAssembler
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline

object Next {
  val spark : SparkSession = SparkSession.builder
    .appName("spark-nlp-starter")
    .master("local[*]")
    .getOrCreate

  val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
  val database_name = "Spark_base"
  val url = server_name + ";" + "databaseName=" + database_name + ";"
  val table_name = "man_in_the_case"
  val username = "sa"
  val password = "user"
  val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"

  import spark.implicits._

  def main(args: Array[String]): Unit = {
    val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"
    val textData = spark.read.textFile(textFile)
    val CleanData = textData
        .filter(line => !line.isEmpty).map(line => line.trim())

    val DocumentAssembler = new DocumentAssembler()
        .setInputCol("value")
        .setOutputCol("document")
```

```
val sentenceDetector = new SentenceDetector()
  .setInputCols("document")
  .setOutputCol("sentence")

val token = new Tokenizer()
  .setInputCols("sentence")
  .setOutputCol("token")

val normalizer = new annotators.Normalizer()
  .setInputCols("token")
  .setOutputCol("normalized")
  .setLowercase(true)
  .setCleanupPatterns(Array("""[^\w\d\s]"""))

val p = new Pipeline()
    .setStages(Array(
     DocumentAssembler,
     sentenceDetector,
     token,
     normalizer))

val result = p.fit(CleanData).transform(CleanData)

val fin = result.select("normalized.result")
  .withColumn("result", explode($"result"))
  .map(line => line.getString(0).trim)

val hund = fin.count().toFloat

val dif_w_count = fin.groupBy("value").count()

val col_per = dif_w_count.select("count")
  .map(x => ((x.getLong(0).toFloat / hund) * 100f))
```

```
      .withColumn("id", monotonically_increasing_id())

    val word_frequency = dif_w_count.select("value")
      .withColumn("id", monotonically_increasing_id)
      .join(col_per, Seq("id")).drop("id")
      .toDF(Seq("Words", "Percents" ) : _*)
    word_frequency.sort(desc("Percents")).show()

    word_frequency.write
      .format("jdbc")
      .mode("overwrite") /*overwrite*/
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", table_name)
      .option("user", username)
      .option("password", password)
      .save()
  }
}
```

Пятый пример демонстрирует использование Pos теггера. Тут как и в прошлом примере будут инициализированны модели DocumentAssembler, SentenceDetector, Tokenizer, Normalizer. В добавление к ним будет инициализирован posTegger. Для этого была загружена предварительно обученная модель с сайта `https://nlp.johnsnowlabs.com/models`. Инициализируется с помощью функции load, которая содержит путь в котором хранятся данные для инициализации и работы модели. Эта модель принимает на вход столбцы document и normalizer (так же можно token), и присваивает каждому токену аргумент, POS tag. Они имеют различные значения указывающие на принадлежность элемента к части речи естественного языка. После чего формируется пайплайн на основе, определенных выше моделей. После чего датафрейм трансофрмируется, с помощью определенного пайплайна. К датафрейму применяется функция groupBy, после чего применяется функция count, в результате чего формируется датафрейм содержащий уникальные POS теги и количество уникальных тегов. Полученный датафрейм загружается в SQL

Server.

```
import com.johnsnowlabs.nlp.annotator._
import org.apache.spark.sql.functions._
import com.johnsnowlabs.nlp.{SparkNLP, annotators}
import com.johnsnowlabs.nlp.base.DocumentAssembler
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline

object Next {
  val spark : SparkSession = SparkSession.builder
    .appName("spark-nlp-starter")
    .master("local[*]")
    .getOrCreate

  val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
  val database_name = "Spark_base"
  val url = server_name + ";" + "databaseName=" + database_name + ";"
  val table_name = "man_in_the_case"
  val username = "sa"
  val password = "user"
  val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"

  import spark.implicits._

  def main(args: Array[String]): Unit = {
    val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"
    val textData = spark.read.textFile(textFile)
    val CleanData = textData
        .filter(line => !line.isEmpty).map(line => line.trim())

    val DocumentAssembler = new DocumentAssembler()
        .setInputCol("value")
        .setOutputCol("document")
```

```
val sentenceDetector = new SentenceDetector()
  .setInputCols("document")
  .setOutputCol("sentence")

val token = new Tokenizer()
  .setInputCols("sentence")
  .setOutputCol("token")

val normalizer = new annotators.Normalizer()
  .setInputCols("token")
  .setOutputCol("normalized")
  .setLowercase(true)
  .setCleanupPatterns(Array("""[^\w\d\s]"""))

val posTagger = PerceptronModel.read
  .load("pos_ud_ewt_en_3.0.0_3.0_1615230175426/")
  .setInputCols("document","normalized")
  .setOutputCol("pos")

val p_tag = new Pipeline()
  .setStages(Array(
    DocumentAssembler,
    sentenceDetector,
    token,
    normalizer,
    posTagger))

val pos_df = p_tag.fit(CleanData)
  .transform(CleanData).select("pos.result")
  .withColumn("result", explode($"result"))
  .map(line => line.getString(0).trim)

val pos_table = pos_df.groupBy("value").count()
```

```
    pos_table.show()

    pos_table.write
      .format("jdbc")
      .mode("overwrite") /*overwrite*/
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", "pos_man_in_case")
      .option("user", username)
      .option("password", password)
      .save()
}
}
```

3.2 Оформление отчетов с помощью PowerBI

Power Bi не может забрать данные напрямую из Spark, поэтому нам и нужны таблицы базы данных SQLServer. Что бы интегрировать данные из базы данных, нужно просто в меню Microsoft PowerBi, выбрать пункт |Get data| (см. рис 4).

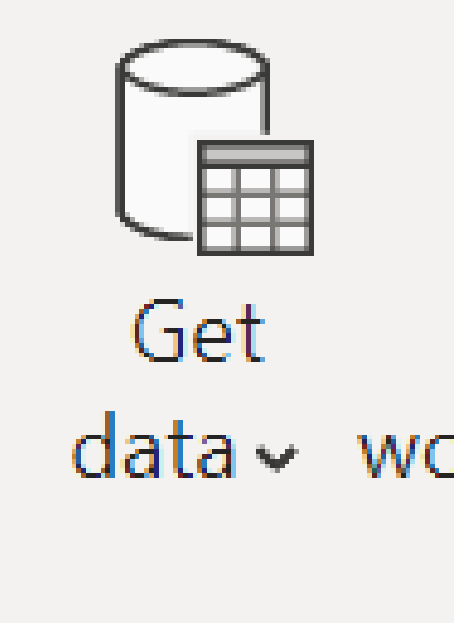


Рисунок 4 – Подпись к рисунку

После чего необходимо ввести имя сервера, и выбрать базу данных и таблицы данные которых мы хотим загрузить (см. рисунки 5, 6).

Рисунок 5 – Подпись к рисунку

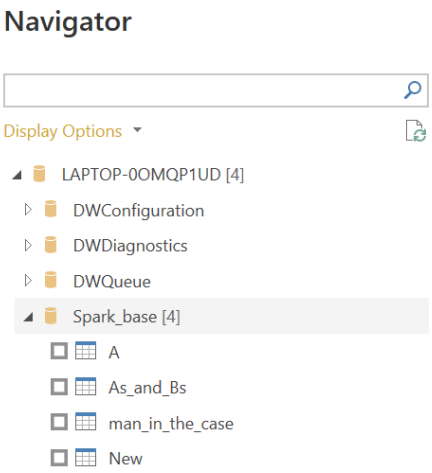


Рисунок 6 – Подпись к рисунку

Теперь когда данные загружены, можно составить отчет на основе полученных данных. Для этого я выбрал несколько визуализаций,и построил их на примере некоторых полученных мною таблиц (см. рисунки 7,8,9,10,11,12).

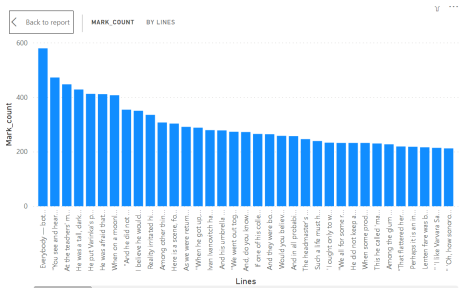


Рисунок 7 – Диаграмма отображающая количество символов к строке их содержащую

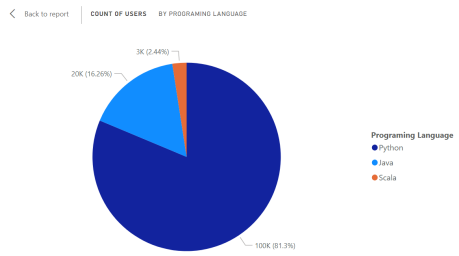


Рисунок 8 – Круговая диаграмма,основанная на таблице из первого примера

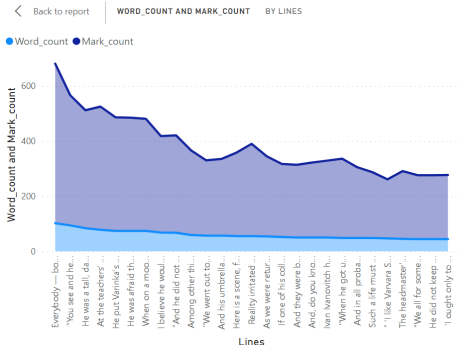


Рисунок 9 – Диаграмма областей,основанная на третьем примере

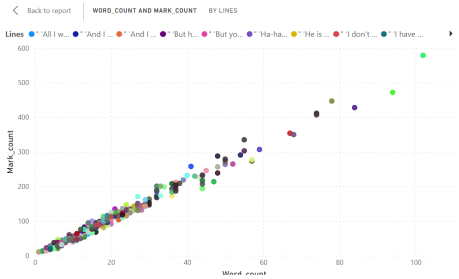


Рисунок 10 – Диаграмма рассеяния слов и знаков содержащихся в строке

Рисунок 11 – Древовидная карта частоты употребления слов в тексте

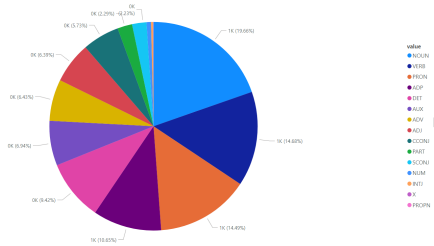


Рисунок 12 – Круговая диаграмма распределения частей речи в тексте

ЗАКЛЮЧЕНИЕ

В настоящей работе были решены следующие задачи:

- Изучено взаимодействия Apache Spark и SQL Server.
- Реализовано несколько примеров ,основанных на анализе неструктурированных данных и загрузки результатов обработки в SQL Server.

SQL Server, отлично подходит для хранения результатов обработки , так как в Spark данные не обновляемые они считываются и записываются один раз, тогда как данные в SQL Server, можно перезаписывать и обновлять,что может пригодится для некоторых задач.Так же при обработке SQL Server поддерживает принципы ACID, чего не хватает NoSQL решениям. Кроме того SQL Server поддерживает инструменты оформления отчетов, что является полезным в сфере обработки данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Технологии Big Data: как использовать большие данные в маркетинге [Электронный ресурс]. — URL: https://www.uplab.ru/blog/big-data-technologies/(Дата обращения 28.9.2021). Загл. с экр. Яз. рус.

2 Аналитический обзор рынка Big Data // Хабрахабр [Электронный ресурс]. — URL: https://habrahabr.ru/company/moex/blog/256747/ (Дата обращения 30.9.2021). Загл. с экр. Яз. рус.

3 Big Data: что это и где применяется? [Электронный ресурс]. — URL: https://blog.skillfactory.ru/chto-takoe-bolshie-dannye/(Дата обращения 1.10.2021). Загл. с экр. Яз. рус.

4 Как работают реляционные БД [Электронный ресурс]. — URL: https://habr.com/ru/company/vk/blog/266811/(Дата обращения 11.10.2021). Загл. с экр. Яз. рус.

5 NoSQL базы данных: понимаем суть [Электронный ресурс]. — URL: https://habr.com/ru/post/152477/(Дата обращения 10.10.2021). Загл. с экр. Яз. рус.

6 Базы данных SQL, NoSQL и различия в моделях баз данных [Электронный ресурс]. — URL: https://devacademy.ru/article/sql-nosql/(Дата обращения 11.10.2021). Загл. с экр. Яз. рус.

7 CAP и BASE [Электронный ресурс]. — URL: https://russianblogs.com/article/3841900241/(Дата обращения 12.10.2021). Загл. с экр. Яз. рус.

8 NoSQL базы данных [Электронный ресурс]. — URL: https://cloud.yandex.ru/promo/nosql-databases/(Дата обращения 12.10.2021). Загл. с экр. Яз. рус.

9 Что такое базы данных NoSQL? [Электронный ресурс]. — URL: https://azure.microsoft.com/ru-ru/overview/nosql-database/(Дата обращения 19.10.2021). Загл. с экр. Яз. рус.

10 SQL или NoSQL [Электронный ресурс]. — URL: https://habr.com/ru/company/ruvds/blog/324936/(Дата обращения 23.10.2021). Загл. с экр. Яз. рус.

11 Анализ проблем безопасности архитектуры распределённых NoSQL приложений на примере программного каркаса Hadoop // Портал магистров ДонНТУ. [Электронный ресурс]. — URL: http://masters.donntu.org/2014/fknt/chuprin/library/_hadoop-security.htm\T2A\cyrd (Дата обращения 30.9.2021). Загл. с экр. Яз. рус.

12 A Survey on Data Security System for Cloud Using Hadoop // International Journal of Innovative Research in Computer and Communication Engineering. [Электронный ресурс]. — URL: https://www.ijircce.com/upload/2016/november/164_A%20SURVEY.pdf (Дата обращения 23.10.2021). Загл. с экр. Яз. рус.

13 Introduction to MapReduce [Электронный ресурс]. — URL: http://sci2s.ugr.es/BigData#Big%20Data%20Technologies (Дата обращения 24.10.2021). Загл. с экр. Яз. рус.

14 *Крау, Х.* Изучаем Spark. Молниеносный анализ данных. / Х. Крау. — БХВ - Петербург,2022, 2015. — С. 18 – 21.

15 How to install ApacheSpark on Windows 10 [Электронный ресурс]. — URL: http://msdn.microsoft.com/ru-ru/library/vstudio/ms235634(v=vs.100).aspx (Дата обращения 26.10.2021). Загл. с экр. Яз. рус.

16 Apache spark: Что там под копотом? [Электронный ресурс]. — URL: https://habr.com/ru/post/251507/ (Дата обращения 27.10.2021). Загл. с экр. Яз. рус.

17 Class DataFrame [Электронный ресурс]. — URL: https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/sql/DataFrame.html (Дата обращения 29.10.2021). Загл. с экр. Яз. рус.

18 Компоненты SparkNLP [Электронный ресурс]. — URL: https://spark-school.ru/blogs/components-spark-nlp/ (Дата обращения 29.10.2021). Загл. с экр. Яз. рус.

19 Введение в НЛП Spark: Основы и основные компоненты [Электронный ресурс]. — URL: https://towardsdatascience.com/introduction-to-spark-nlp-foundations-and-basic-components-part-i-c8gi=e4c4d93693c8 (Дата обращения 30.10.2021). Загл. с экр. Яз. рус.

20 Apache Spark connector : SQL Server and Azure SQL [Электронный ресурс]. — URL: https://docs.microsoft.com/en-us/sql/connect/spark/connector?view=sql-server-ver15 (Дата обращения 30.10.2021). Загл. с экр. Яз. рус.

21 Microsoft SQL Server [Электронный ресурс]. — URL: http://bourabai.kz/dbt/servers/MicrosoftSQLServer.htm(Дата обращения 30.10.2021). Загл. с экр. Яз. рус.

22 *Молинаро, Э.* SQL Сборник рецептов. – 2е издание / Э. Молинаро. — БХВ - Петербург, 2022, 2021. — С. 30.

23 Руководство по PowerBi [Электронный ресурс]. — URL: https://habr.com/ru/company/microsoft/blog/427701/ (Дата обращения 30.10.2021). Загл. с экр. Яз. рус.

ПРИЛОЖЕНИЕ А

Листинг первого примера

```
import org.apache.spark.sql.SparkSession

object Connector_demo {
    def main (args: Array[String]) :Unit = {
      val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
      val database_name = "Spark_base"
      val url =
            server_name + ";" + "databaseName=" + database_name + ";"

      val table_name = "New"
      val username = "sa"
      val password = "user"
      val logFile =
                "C:/MyFiles/Spark/spark-3.1.2-bin-hadoop3.2/README.md"

      val spark = SparkSession.builder.appName("Simple Application")
                                        .getOrCreate()
      val columns = Seq("language","users_count")
      val data = Seq(("Java", "20000"),
                     ("Python", "100000"),
                     ("Scala", "3000"))
      val rdd = spark.sparkContext.parallelize(data)
      val df1 = spark.createDataFrame(rdd)
      val dfRe = df1.toDF(columns : _*)
      dfRe.show()
      dfRe.write
      .format("jdbc")
      .mode("overwrite")
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", table_name)
      .option("user", username)
```

```
        .option("password", password)
        .save()
    }
}
```

ПРИЛОЖЕНИЕ Б

Листинг второго примера

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{monotonically_increasing_id}
object A_B {
    def main (args: Array[String]) : Unit = {
      val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
      val database_name = "Spark_base"
      val url =
            server_name + ";" + "databaseName=" + database_name + ";"
      val table_name = "As_and_Bs"
      val username = "sa"
      val password = "user"
      val logFile =
              "C:/MyFiles/Spark/spark-3.1.2-bin-hadoop3.2/README.md"

      val spark = SparkSession.builder.appName("A_B").getOrCreate()
      val logData = spark.read.textFile(logFile).cache()
      val As = logData.filter(line => line.contains("a"))
      val Bs = logData.filter(line => line.contains("b"))
      val A_1 = As.withColumn("id", monotonically_increasing_id)
      val B_1 = Bs.withColumn("id", monotonically_increasing_id)
      val data = A_1.join(B_1, Seq("id")).drop("id")
      val columns = Seq("A_word_line", "B_word_line")
      val dfRe = data.toDF(columns : _*)
      dfRe.write
        .format("jdbc")
        .mode("overwrite") /*overwrite*/
        .option("driver",
                "com.microsoft.sqlserver.jdbc.SQLServerDriver")
        .option("url", url)
        .option("dbtable", table_name)
        .option("user", username)
        .option("password", password)
```

```
            .save()
        spark.stop()

    }
}
```

ПРИЛОЖЕНИЕ В

Листинг третьего примера

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, explode, monotonically_incre
object Text_file {

  def main (args: Array[String]) :Unit = {

    val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
    val database_name = "Spark_base"
    val url =
           server_name + ";" + "databaseName=" + database_name + ";"

    val table_name = "man_in_the_case"
    val username = "sa"
    val password = "user"
    val textFile =
                "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"

    val spark = SparkSession.builder.appName("text_app").getOrCreate()
    import spark.implicits._
    import spark.createDataFrame
    val textData = spark.read.textFile(textFile)
    val CleanData = textData.filter(line => !line.isEmpty)
    val lines = CleanData.withColumn("id", monotonically_increasing_id)

    val m_c = CleanData.map(lines => lines.replace(" ", "").length)
    val mark_count = m_c.withColumn("id", monotonically_increasing_id)

    val words = CleanData.map(lines => lines.replaceAll("\\pP", "")
                             .split(" "))
    val w_c = words.map(lines => lines.length)
    val word_count = w_c.withColumn("id", monotonically_increasing_id)
```

```
    val data = lines.join(mark_count, Seq("id"))
                        .join(word_count, Seq("id")).drop("id")
    val columns = Seq("Lines", "Mark_count", "Word_count")
    val dfRe = data.toDF(columns : _*)
    dfRe.show()
    dfRe.write
      .format("jdbc")
      .mode("overwrite")
      .option("driver",
              "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", table_name)
      .option("user", username)
      .option("password", password)
      .save()
  }
}
```

ПРИЛОЖЕНИЕ Г

Листинг четвертого примера

```
import com.johnsnowlabs.nlp.annotator._
import org.apache.spark.sql.functions._
import com.johnsnowlabs.nlp.{SparkNLP, annotators}
import com.johnsnowlabs.nlp.base.DocumentAssembler
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline

object Next {
  val spark : SparkSession = SparkSession.builder
    .appName("spark-nlp-starter")
    .master("local[*]")
    .getOrCreate

  val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
  val database_name = "Spark_base"
  val url = server_name + ";" + "databaseName=" + database_name + ";"
  val table_name = "man_in_the_case"
  val username = "sa"
  val password = "user"
  val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"

  import spark.implicits._

  def main(args: Array[String]): Unit = {
    val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"
    val textData = spark.read.textFile(textFile)
    val CleanData = textData
        .filter(line => !line.isEmpty).map(line => line.trim())

    val DocumentAssembler = new DocumentAssembler()
        .setInputCol("value")
        .setOutputCol("document")
```

```
val sentenceDetector = new SentenceDetector()
  .setInputCols("document")
  .setOutputCol("sentence")

val token = new Tokenizer()
  .setInputCols("sentence")
  .setOutputCol("token")

val normalizer = new annotators.Normalizer()
  .setInputCols("token")
  .setOutputCol("normalized")
  .setLowercase(true)
  .setCleanupPatterns(Array("""[^\w\d\s]"""))

val p = new Pipeline()
    .setStages(Array(
     DocumentAssembler,
     sentenceDetector,
     token,
     normalizer))

val result = p.fit(CleanData).transform(CleanData)

val fin = result.select("normalized.result")
  .withColumn("result", explode($"result"))
  .map(line => line.getString(0).trim)

val hund = fin.count().toFloat

val dif_w_count = fin.groupBy("value").count()

val col_per = dif_w_count.select("count")
  .map(x => ((x.getLong(0).toFloat / hund) * 100f))
```

```
      .withColumn("id", monotonically_increasing_id())

    val word_frequency = dif_w_count.select("value")
      .withColumn("id", monotonically_increasing_id)
      .join(col_per, Seq("id")).drop("id")
      .toDF(Seq("Words", "Percents" ) : _*)
    word_frequency.sort(desc("Percents")).show()

    word_frequency.write
      .format("jdbc")
      .mode("overwrite") /*overwrite*/
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", table_name)
      .option("user", username)
      .option("password", password)
      .save()
  }
}
```

ПРИЛОЖЕНИЕ Д

Листинг пятого примера

```
import com.johnsnowlabs.nlp.annotator._
import org.apache.spark.sql.functions._
import com.johnsnowlabs.nlp.{SparkNLP, annotators}
import com.johnsnowlabs.nlp.base.DocumentAssembler
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline

object Next {
  val spark : SparkSession = SparkSession.builder
    .appName("spark-nlp-starter")
    .master("local[*]")
    .getOrCreate

  val server_name = "jdbc:sqlserver://LAPTOP-0OMQP1UD"
  val database_name = "Spark_base"
  val url = server_name + ";" + "databaseName=" + database_name + ";"
  val table_name = "man_in_the_case"
  val username = "sa"
  val password = "user"
  val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"

  import spark.implicits._

  def main(args: Array[String]): Unit = {
    val textFile = "C:/MyFiles/Text_Sampels/the_Man_in_the_Case.txt"
    val textData = spark.read.textFile(textFile)
    val CleanData = textData
        .filter(line => !line.isEmpty).map(line => line.trim())

    val DocumentAssembler = new DocumentAssembler()
        .setInputCol("value")
        .setOutputCol("document")
```

```
val sentenceDetector = new SentenceDetector()
  .setInputCols("document")
  .setOutputCol("sentence")

val token = new Tokenizer()
  .setInputCols("sentence")
  .setOutputCol("token")

val normalizer = new annotators.Normalizer()
  .setInputCols("token")
  .setOutputCol("normalized")
  .setLowercase(true)
  .setCleanupPatterns(Array("""[^\w\d\s]"""))

val posTagger = PerceptronModel.read
  .load("pos_ud_ewt_en_3.0.0_3.0_1615230175426/")
  .setInputCols("document","normalized")
  .setOutputCol("pos")

val p_tag = new Pipeline()
  .setStages(Array(
    DocumentAssembler,
    sentenceDetector,
    token,
    normalizer,
    posTagger))

val pos_df = p_tag.fit(CleanData)
  .transform(CleanData).select("pos.result")
  .withColumn("result", explode($"result"))
  .map(line => line.getString(0).trim)

val pos_table = pos_df.groupBy("value").count()
```

```
    pos_table.show()

    pos_table.write
      .format("jdbc")
      .mode("overwrite") /*overwrite*/
      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
      .option("url", url)
      .option("dbtable", "pos_man_in_case")
      .option("user", username)
      .option("password", password)
      .save()
}
}
```